Naval Research Laboratory

Washington, DC 20375-5320

NRL/MR/5510--96-7888

Evolving Fuzzy Logic Control Strategies using SAMUEL: An Initial Implementation

HELEN G. COBB
JOHN J. GREFENSTETTE

Navy Center for Applied Research in Artificial Intelligence
Information Technology Division

September 6, 1996

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave Blank)*	2. REPORT DATE September 6, 1996	3. REPORT TYPE AND DATES COVERED

4. TITLE AND SUBTITLE

Evolving Fuzzy Logic Control Strategies using SAMUEL: An Initial Implementation

5. FUNDING NUMBERS

PN - 55-6471
PE - 62234N
TA - 129

6. AUTHOR(S)

Helen G. Cobb and John J. Grefenstette

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Naval Research Laboratory
Washington, DC 20375-5320

8. PERFORMING ORGANIZATION REPORT NUMBER

NRL/MR/5510--96-7888

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Office of Naval Research
800 North Quincy Street
Arlington, VA 22217-5660

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

Many control systems have been successfully implemented using fuzzy logic, which provides a systematic method for reasoning about uncertainty using expressions found in natural language. This paper describes an extension of the SAMUEL learning system to include fuzzy logic. SAMUEL is a learning system that uses genetic algorithms and other learning methods to evolve refined rules from an initial set of rules provided by the user. In this initial implementation, SAMUEL searches for the rules making up a fuzzy knowledge base (that is, a control strategy), given the user's definition of the fuzzy variables, the values that the variables can take on, and the fixed membership functions associated with the fuzzy values. The genetic algorithm searches for the combinations of rules that make up effective strategies, including the level of generality expressed by the rules. An example is provided showing how to learn fuzzy rules for evasive maneuvers.

14. SUBJECT TERMS

Fuzzy Logic
Machine learning
Genetic algorithms

15. NUMBER OF PAGES

40

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std 239-18
298-102

CONTENTS

EVOLVING FUZZY LOGIC CONTROL STRATEGIES USING SAMUEL: AN INITIAL IMPLEMENTATION

INTRODUCTION

In the past several years, there has been a growing interest in fuzzy logic. Fuzzy logic set theory draws on earlier work conducted in the 1930s: the continuous-valued logic of logician Jan Lukasiewicz and the set theoretics as applied to continuous-valued logic of quantum philosopher (Kosko, 1991). Lotfi Zadeh's seminal paper, published in 1965, extends this work to develop the formal fuzzy-set theory that is used today (Lee, 1990). Unlike Boolean logic, in which membership in a set only takes on the two values of *true* and *false*, in fuzzy logic, membership is multivalued. The degree to which an object belongs to a fuzzy set is expressed using a multivalued indicator called a *membership function*. Fuzzy logic encompasses two-valued logic as a special case.

There are some advantages to using fuzzy logic. One of the advantages is that variables can be expressed linguistically. Fuzzy logic provides a systematic method for reasoning about uncertainty using many of the expressions found in natural language. Conditions within if-then rules are expressed using fuzzy variables that take on imprecise *linguistic values*. For example, the variable *distance* might take on values consisting of quantifiers such as *far*, *close*, and *near*, modified by predicates such as *very*, *not*, and *more or less*. Another advantage is that fuzzy logic provides "natural interpolation and generalization between rules for input/output situations that are not explicitly represented in the rules" (Harris, Moore, and Brown, 1993, pp. 19). With only a limited number of rules, a non-fuzzy system may produce a default output that creates instabilities whenever the system fails to recognize an input, instead of producing an output that is at least approximately correct.

Due to these advantages, many control systems have been successfully implemented using fuzzy logic controllers (FLCs). FLCs have been developed for applications "in fields ranging from consumer electronics and photography to medical-diagnosis systems and securities-management" (Zadeh, 1992, pp. 23). The successful research and development of these fuzzy logic applications in Japan over the last decade has recently encouraged a growing interest in fuzzy logic in the United States (Lee, 1990; Walter, 1993). A good overview of fuzzy logic and its application to control can be found in (Lee, 1990).

Despite the success of fuzzy logic applications, "at present there is no systematic procedure for the design of an [sic] FLC" (Lee, 1990, pp. 404). To define the membership functions used in fuzzy logic applications, an expert is usually required. Many researchers report difficulty in manually constructing fuzzy logic control rules, especially for problem domains where there is no expert knowledge (e.g., Lee and Takagi, 1993; Karr, 1991). A basic approach for overcoming this problem is to use hybrid systems. For example, neural networks have been used to develop or fine-tune fuzzy logic knowledge based systems (Takagi, 1993). More recently, genetic algorithms (GAs) have been used to automate the search (Sun and Jang, 1993; Karr, 1991; Thrift, 1991). One group of researchers have explored using a GA to develop a fuzzy knowledge-based system that controls the parameters settings of another GA (Lee and Takagi, 1993). Most of this research indicates that there is promise in combining genetic algorithm-based machine learning with fuzzy logic.

Manuscript approved July 23, 1996.

To design a FLC, there are four major considerations: (1) the number of fuzzy rules, (2) the shape of the membership functions, (3) the fuzzy input variables and control actions (rule antecedents and consequents), and (4) the reasoning method (Takagi, 1993). In most genetic algorithm research to date, the GA searches either for the parameters settings that describe membership functions, or for the fuzzy rules making up the rulebase, once the fuzzy variables have been defined and the number of rules in the controller has been decided (Thrift, 1991). This report addresses the initial implementation of fuzzy logic control in SAMUEL, a genetic algorithm-based machine learning system developed at the Naval Research Laboratory's Navy Center for Applied Research Artificial Intelligence (NCARAI) (Grefenstette, 1988). In this implementation, SAMUEL searches for the rules making up the fuzzy knowledge base (i.e., a control strategy), given a user's definition of the fuzzy variables, the values that the variables can take on, and the fixed membership functions associated with the fuzzy values.

SAMUEL uses competitive learning to evolve control strategies that consist of sets of if-then production rules. There are two levels of competition in the system. At the lower level, rules within a strategy compete with one another to determine the current control action. Rules bid to fire based on their degree of match with the incoming sensory inputs and on the rules' relative strengths. The Competitive Production System (CPS) in SAMUEL performs credit assignment at the end of a learning episode to adjust the strengths of rules, depending on the overall payoff of the episode. Thus CPS represents the reinforcement learning part of SAMUEL. Averages of the payoff values from a number of learning episodes are used to compute the overall fitness of a control strategy. At the higher level in SAMUEL, the control strategies making up the population in the GA compete with one another in order to determine which strategies will contribute to the next generation's population. The GA uses CPS's evaluations to select strategies for reproduction. Successful strategies are mated by recombining their rules to form new strategies. In addition, new rules may be introduced into strategies by applying various mutation operators (e.g., CREEP) and modification operators (e.g., SPECIALIZE and GENERALIZE) to predecessor rules. The implicitly parallel search of the GA supports the search of the complex space of control strategies. For more information about the philosophy behind SAMUEL and a more extensive explanation of the system's operation, please see the *User's Guide for SAMUEL, Version 4* (Grefenstette and Cobb, 1995) and the research papers that describe experiments using the system (e.g., Grefenstette, 1988; Grefenstette, Ramsey and Schultz, 1990; Grefenstette, 1991). The *User's Guide* also includes a discussion on how fuzzy logic is incorporated into SAMUEL.

SAMUEL has several characteristics that make it compatible with a fuzzy logic approach to control. One compatibility is that SAMUEL supports to a large extent the kind of linguistic knowledge bases that are typical of fuzzy logic controllers. Another compatibility is that SAMUEL uses a simple inference mechanism typical of FLCs where "the consequent of a rule is not applied to the antecedent of another. In other words, in FLC[s] we do not employ the chaining inference mechanism, since the control actions are based on one-level forward data-driven inference (GMP)" (Lee, 1990, pp.425). In addition, the inference mechanism is compatible in that it already uses partial rule matching. Partial matching effectively smooths the operating regions of the control space by considering each rule's degree of match with sensory inputs. While different in detail, this partial matching mechanism is similar in spirit to fuzzy logic. All of these compatibilities suggest that fuzzy logic control could be added to SAMUEL's current capabilities without changing the design philosophy behind the system.

Like many genetic algorithm-based systems, SAMUEL requires a minimum of external background knowledge to begin its search for successful control strategies, although the system has the flexibility of incorporating an expert's knowledge. Rules are expressed in a representational language that can be easily specified and interpreted by a user of the system. A user can express the attributes of rule conditions either as numeric ranges or in terms of more linguistically structured hierarchical relationships. Rules where conditions (i.e., fuzzy variables) take on linguistic values are supported using the rule syntax of tree *structured* *attributes*. Thus, many of the control strategies that SAMUEL currently evolves consist of rules that are similar in appearance to the productions found in FLCs. The only modification required to support the rule syntax of FLCs is the specification of membership functions at the leaf nodes of the structured attribute trees.

Another modification that is required in order to incorporate fuzzy logic into SAMUEL is providing a fuzzy logic inference mechanism as an alternative to the system's default rule matching mechanism. SAMUEL's rule matching and bidding mechanisms combined ultimately determine the agent's control action. In typical non-fuzzy controllers, the control space is partitioned into non-overlapping regions. The action that an agent takes is determined solely by how the current sensory inputs map into one of these partitions. In FLCs, each partition contributes to the control decision to the extent that the inputs belong to the partition (Lee and Takagi, 1993, pp. 77). Interestingly, both SAMUEL's rule matching mechanism and fuzzy logic mechanisms select a rule to fire based on its partial match with the current sensory inputs. The default matching mechanism in SAMUEL measures partial match in terms of the number of conditions in a rule that match the current sensory readings. In contrast, FLCs measure partial match for a rule's conditions based on the inputs' level of membership over those conditions. For example, a standard technique is to find the minimum membership value over the conditions, which effectively computes the intersection of the partial matches. Because both mechanisms perform partial matching in some sense, it seems natural to expand SAMUEL's capabilities to include an option for using fuzzy logic inference.

SAMUEL differs, however, from a standard (i.e., non-adaptive) FLC in that the system incorporates learned information in its inference mechanism. Recall that after rule matching, rules enter a bidding process that considers the rules' strengths as well as the degree of partial matching to select a control action. These bids effectively weight the relative importance of the rules to reflect the learner's experience. In FLCs, all of the rules in the knowledge base that have the same degree of match typically receive equal consideration in determining the control response.

In the current implementation of fuzzy logic in SAMUEL, the membership functions of the fuzzy values remain the same for all strategies considered by the GA. The GA searches for the combinations of rules that make up effective strategies, including the level of generality expressed by the rules; CPS learns the strength of the rules within a strategy, which affects how SAMUEL performs fuzzy inference. Future work will address methods for learning membership functions as well as combinations of rules.

The remainder of the report is organized as follows. The *Overview of Fuzzy Logic* section provides enough of a brief overview of fuzzy logic. The section, *Implementation of Fuzzy Logic in SAMUEL*, discusses specific details, including how to specify fuzzy variables as conditions in the system's attributes file, and a description of the procedures the system uses to perform fuzzy logic inference. The subsequent section, *The Evade Problem: An Example*, provides an illustration of how to run the system to develop a fuzzy logic controller. Finally, the last section discusses future enhancements of the system.

OVERVIEW OF FUZZY LOGIC

Fuzzy logic set theory can be thought of as a generalization of ordinary set theory. In ordinary sets, an object either belongs to or does not belong to a set. For this reason, ordinary sets are often called *crisp* sets in fuzzy logic parlance. In fuzzy logic set theory, "the transition from membership to non-membership is gradual rather than abrupt" (Zadeh, 1973, pp. 28) so that an object belongs to a set with some degree of certainty. According to Kosko, this sense of "certainty" differs from the probabilistic notion: "Fuzziness describes *event ambiguity*" (Kosko, 1991, pp. 265). "Ambiguity is a property of physical phenomena" (Kosko, 1991, pp. 267). In contrast: "Randomness describes the uncertainty of *event occurrence*." (Kosko, 1991, pp. 265). "Unlike fuzziness, probability dissipates with increasing information" (Kosko, 1991, pp.

297). Thus, fuzziness is not simply due to lack of information about an event. Zadeh characterizes fuzziness as the result of having to summarize large quantities of data (Zadeh, 1973).

Definitions

The *membership function*, $\mu_A(x)$, characterizes the level of membership an object x has within set A, where the level of membership, or degree of belief concerning membership, is expressed as a value from the closed interval [0, 1]. A larger value expresses more belief in the membership. Formally, $\mu_A(x)$: $U \rightarrow [0, 1]$, for fuzzy set $A \subset U$, where U is the universe of discourse. The *support set* of a fuzzy set A consists of all objects x whose $\mu_A(x) > 0$.

The membership function of the *intersection* of two sets, A and B, having the *same object x* can be interpreted as the *min* operator, or as the product of Larsen operator, *, (Harris, Moore, & Brown, 1993, pp. 12). Typically, the min operator is used:

$$\mu_{A \cap B}(x) = \mu_A(x) \wedge \mu_B(x) = min\{\mu_A(x), \mu_B(x)\} \qquad \text{(Eq 1)}$$

The imprecision of an input x vector to a process can be calculated using the *Cartesian product.* If $\mu_{A_i}(x_i)$ is the belief that x_i is a member of A_i for the ith sensory input, then the membership function for belonging to the set of sensory inputs is

$$\mu_{A_1 \times A_2 \times A_3, \ldots, A_n}(x) = min\{\mu_{A_1}(x_1), \mu_{A_2}(x_2), \mu_{A_3}(x_3), \ldots, \mu_{A_n}(x_n))\} \qquad \text{(Eq 2)}$$

Thus, the membership value for matching the conditions of a rule is the min of the membership values over the conditions. The fuzzy implementation of SAMUEL uses this principle in the procedure *samuel_find_fuzzy_matches()*.

The membership of the *union* of two sets, A and B, having the same object x can be interpreted as the *max* operator, or as the algebraic sum (Harris, Moore, & Brown, 1993, pp. 12). Typically, the max operator is used:

$$\mu_{A \cup B}(x) = \mu_A(x) \vee \mu_B(x) = max\{\mu_A(x), \mu_B(x)\} \qquad \text{(Eq 3)}$$

For example, suppose the sets A and B are leaves in a tree structured hierarchy as shown in Figure 1.

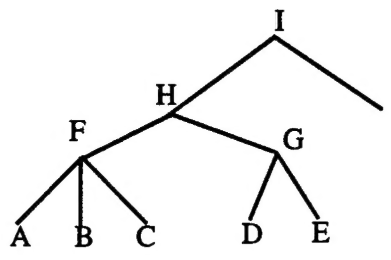

Fig. 1 - Tree Structured Hierarchy

If object x is in the support set of A and B and not in the support set of C, then $\mu_F(x) = \mu_A(x) \vee \mu_B(x)$. The fuzzy implementation of SAMUEL uses conditions having a structured attribute type. Thus SAMUEL's *samuel_fuzzy_match_atom()* computes the degree of match for a condition as the maximum membership value of the leaf nodes covering the input.

Fuzzy Inference

The basic steps in performing fuzzy inference consists of processing inputs through a *fuzzification interface*, applying *decision-making logic* to a fuzzy knowledge base, and converting fuzzy outputs into crisp ones using a *defuzzification interface* (Lee, 1990, pp. 406). The major function of the fuzzification interface is to convert sensory data into linguistic values. SAMUEL's current sensor interface already performs this function.

The fuzzy knowledge base is a strategy consisting of a set of control rules (i.e., a fuzzy control strategy). For example, a fuzzy control strategy might be

```
R1:    IF speed = slow
       AND time = beginning
       THEN SET turn = hard-right

R2:    IF speed = medium-fast
       AND time = beginning
       THEN SET turn = med-left

...

Rn:    IF speed = medium-slow
       AND time =ending
       THEN SET turn = soft-right
```

In this example, the conditions speed and time are the variables that take on linguistic values (i.e., linguistic variables). Each linguistic value has a membership function associated with it. The rules in the strategy can be thought of as being symbolically connected through the connective *also* (Lee, 1990, pp. 407):

$$R = also\,(R_1, R_2, \ldots, R_i, \ldots, R_n) \qquad \text{(Eq 4)}$$

The decision making logic infers what control action should be taken using the rules of inference in fuzzy logic. One standard technique for deciding among the rules is to consider the rule having the maximum of the membership values since the *also* connective can be interpreted as the union of the rules (Lee, 1990). In current implementation of fuzzy logic in SAMUEL, the maximum membership values of the rules are considered indirectly. Procedure samuel_find_fuzzy_matches() generates match lists which rank the rules based on their membership values so that higher membership values correspond to top ranking match lists. SAMUEL considers the top ranking match list when selecting the value of a control action.

The defuzzification interface converts the fuzzy control output of a rule into a crisp action value. There are two common approaches for defuzzying actions: the Mean of Maximum (MOM) Method, and the Center of Area (COA) Method. In the MOM method, the control action value is the mean of all of the action values having the maximum membership value. In the COA method, membership values are used as weights in forming a weighted average of the action values. Currently, SAMUEL does not defuzzify action values; only one action value is suggested by each rule.

IMPLEMENTATION OF FUZZY LOGIC IN SAMUEL

Membership Functions as Rule Attributes

In order for SAMUEL to perform fuzzy inference, the conditions of rules must be expressed as structured attributes. For structured attributes, each atom in the attribute tree corresponds to a fuzzy variable name. The leaf values of the speed condition consist of the atoms *stopped, slow, medium-slow, medium-paced, medium-fast, fast,* and *full*. The leaves *stopped, slow,* and *medium-slow* are the children of the atom *low*; the leaves *medium-slow, medium-paced, medium-fast* are the children of the atom *medium*; the leaves *medium-fast, fast,* and *full* are the children of the atom *high.*

When a structured attribute specifies *fuzzy matching*, each leaf atom in the structure has a membership function associated with it. For example, Figure 2 shows the membership functions for the speed condition defined as a structured attribute with fuzzy matching. Membership values, μ_{speed}, take on the values between zero and one.

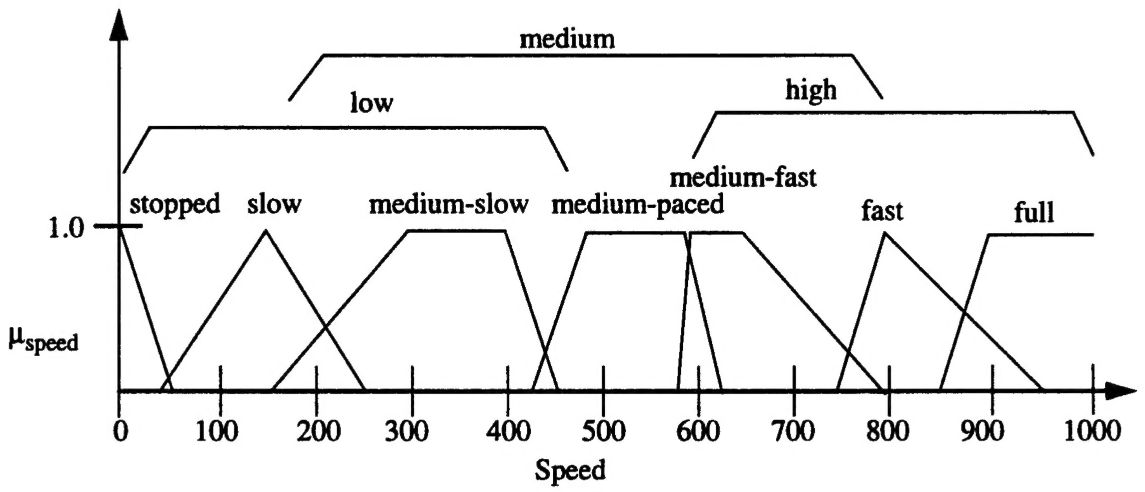

Fig. 2 - Membership functions for the speed condition

Specifying Membership Functions in the attributes File

SAMUEL specifies a learning agent's control strategy using a rulebase. A rulebase may also be used to specify the fixed strategy of a non-learning adversary or the environment. SAMUEL initializes a rulebase using the rule syntax specified in the *attributes* file for that rulebase. If SAMUEL is learning fuzzy logic control strategies, then all of the conditions in the associated attributes file need to be specified as a structured type with matching set to fuzzy.

As an example, consider the specification of the speed condition depicted in Figure 2 as it would be found in an *attributes* file. Table 1 illustrates the speed condition (condition 5) as it might be found in the attributes file. The *type* of the attribute is *structured.* The *match* field indicates that *fuzzy* logic is to be used during rule matching. The *low* and *high* value of the condition specify the discourse for the linguistic variable speed (*i.e.*, the fuzzy_low and the fuzzy_high values).

Table 1. Condition 5 attributes

```
condition 5 :
name = speed
type = structured
match = fuzzy
low  = 0
high = 1000
order = linear
leaf-values = 7
stopped  ( 0, 0, 20, 50 )
slow  ( 40, 150, 150, 250 )
medium-slow  ( 150, 300, 400, 450 )
medium-paced  ( 425, 475, 575, 625 )
medium-fast  ( 580, 600, 650, 800 )
fast  ( 750, 800, 800, 950 )
full  ( 850, 900, 1000, 1000 )
interior-values = 3
name = low
children = 3
stopped slow medium-slow
name = medium
children = 3
medium-slow medium-paced medium-fast
name = high
children = 3
medium-fast fast full
exact = 0
```

The user needs to be careful to define the membership function for the linguistic variables so that the range of the function is specified. The range for the entire speed condition also needs to be defined. For attributes having an order that is *linear*, input values that exceed the high limit are mapped into the fuzzy_high value; input values falling below the lower limit are mapped into the fuzzy_low value. For attributes having a *cyclic* order, modulo arithmetic is performed on values falling outside the range of the discourse. Similarly, modulo arithmetic is performed when the input values fall outside of the lower limit.

Following the name of each of the leaf values is a quadtuple, *(low-partial, low-full, high-full, high-partial),* which describes a membership function. Figure 3 depicts how the quadtuple specifies the membership function for the *medium-paced* fuzzy value. In general, the membership value for low-partial and high-partial is zero; the membership over the interval between low-full and high-full is one, and all mem-. bership values between zero and one are calculated using linear interpolation. One should note that whenever low-full and high-full are the same value, the membership function is triangular rather than trapezoidal in shape.

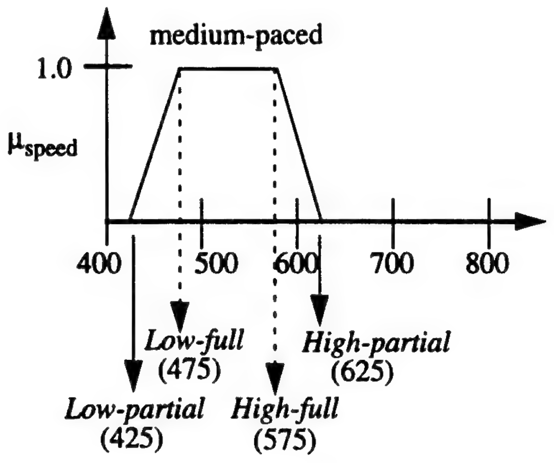

Fig. 3 - Illustration showing how the quadtuple specifies a membership function

Rule Matching

After reading in the rule's attributes, SAMUEL sets the fuzzy flag for the rulebase so that CPS's main procedure, *samuel_cps()*, calls *samuel_find_fuzzy_matches()* instead of the default procedure *samuel_find_matches()* for each agent and for each action an agent can perform. Procedure samuel_find_fuzzy_matches() calls *samuel_fuzzy_match_atom()* for each sensor track, rule number, and condition within a rule to find the membership value for the raw sensor value read in from the agent's sensor track.

To find the maximum membership value, samuel_fuzzy_match_atom() calls three functions: *samuel_fuzzy_no_match()*, *samuel_find_leaf_nodes()*, and *samuel_fuzzy_leaf_match()*. Based on the raw input, samuel_fuzzy_no_match() determines whether the input has a non-zero membership value due to the input lying outside the ranges of the membership functions. If the raw input falls outside the ranges of the membership functions, the samuel_fuzzy_no_match() returns the number of leaf nodes in the attributes' tree. If the number returned is the number of leaf nodes, then this number is returned by samuel_fuzzy_match_atom() along with a membership value of zero. Otherwise, samuel_fuzzy_match_-atom() calls samuel_find_leaf_nodes() to examine the attribute's tree structure among all the nodes that cover the raw sensory input for a condition. Samuel_find_leaf_nodes() returns the number of nodes that cover this sensory input.

For each of node found in samuel_find_leaf_nodes(), samuel_fuzzy_match_atom() calls samuel_-fuzzy_leaf_match() to determine the maximum membership value among the nodes. Depending on which node has the highest value, samuel_fuzzy_match_atom() returns the membership value and the associated linguistic sensor value to samuel_find_fuzzy_matches(). Finally, samuel_find_fuzzy_matches() determines a rule's membership value by finding the minimum membership value among the rule conditions. Appendix B contains a listing of the procedures samuel_find_fuzzy_matches(), samuel_fuzzy_-match_atom(), samuel_fuzzy_no_match(), and samuel_fuzzy_leaf_match().

After finding the minimum membership values among a rule's conditions, samuel_find_fuzzy-_matches() next creates a match list used in the next procedure called by CPS, *samuel_get_bids()* similar to the way *samuel_find_matches()* creates match lists. In samuel_find_matches(), each match list represents the number of conditions that match the incoming sensory inputs. At the extremes, matchlist[conditions + 1][a] saves the rules that match on all conditions for action a, and matchlist[0][a] saves the rules having no matching conditions. Analogously, in samuel_find_fuzzy_matches(), each match list saves rules that cover a range of membership values so that rules are binned into different groups. The width of each range depends on the *Fuzzy_bins* variable, which is set by the *fuzzybins* parameter in the *params* file (the default value is 5 bins). For example, if the Fuzzy_bins is 4, then matchlist[Fuzzy_bins][a] saves those rules whose membership values are less than 1.0 and greater than or equal to 0.75 for action a. At the extremes, matchlist[Fuzzy_bins + 1][a] saves those rules having a membership value of 1.0 for action a (i.e., a perfect match), and matchlist[0][a] saves rules having a membership value of 0.0. Thus, higher indexed match lists, starting with matchlist[Fuzzy_bins + 1][a], store rules having a stronger match with the raw sensory inputs. In both samuel_find_fuzzy_matches() and samuel_find_matches(), the matchcount array saves the number of rules stored in each of the match lists.

Bidding and Conflict Resolution

The fuzzy inference mechanisms in SAMUEL uses the default rule bidding option in samuel_get_bids() (see Grefenstette and Cobb, 1995). For each bidding cycle, one of the rule's control actions is considered. In forming bids, samuel_get_bids() uses the strongest match list having a non-zero match count. This list is called the *match set*. The rules in the match set may suggest different values for the control action. (However, the current implementation assumes that the parameter single_act is set to one so that there is only one action value per rule.) In order to resolve the conflict between competing rules, each action value associated with a rule in the match set bids to be selected. An action value's bid is the strength of the strongest rule in the match set having that action value. Notice that membership values of the rules are not used in the bidding process. Samuel_find_fuzzy_matches() creates match lists based on membership values, and subsequently, samuel_get_bids() selects a rule to fire from among those contained in the best matchlist. This selection is based solely on rule strengths and not membership values.

Next, CPS calls samuel_resolve_conflicts() to determine the winning rule and is action value for the current cycle. Before selecting an action value, bids are modified by raising them to a power that is ten times the value of the *bid bias* parameter. CPS stochastically selects an action value from a probability distribution that is formed using the modified bids. The setting of the bid bias parameter permits a user to shape the bidding distribution. Increasing the bid bias has the effect of emphasizing higher bidding action values over others.

Creating New Rules

In the current version of SAMUEL, the membership functions associated with the leaf atoms do not change. To create new rules using structured attributes, SAMUEL's operators move up and down the levels of an attribute's linguistic tree structure. This traversal has the effect of making the conditions of newly generated rules more general or more specialized than predecessor rules. For example, in creating a new rule, a condition of medium speed in

IF speed = medium
THEN SET turn = hard-right

might be specialized to a medium-slow and medium-paced speed:

IF speed = [medium-slow, medium-paced]
THEN SET turn = hard-right

Similarly, a fast speed could be generalized to high speed, which covers the medium-fast, fast, and full speeds. During matching, the membership values of all a condition's covered nodes are considered. Over the generations, SAMUEL searches for the appropriate level of generality in the rules using the language syntax specified through the attribute structures.

THE EVADE PROBLEM: AN EXAMPLE

The EVADE problem, discussed in Chapters 6 and 7 of the *User's Guide for Samuel, Version 4* (Grefenstette and Cobb, 1995) is used to test SAMUEL's ability to find fuzzy logic control strategies. Please see the *User's Guide* for a more detailed description of this problem.

EVADE is a predator-prey scenario when the prey is the learning agent. The learning objective of the prey is to control its turning rate so that it evades capture by the predator. Each learning episode begins with the predator pursing the prey. The episode ends when (1) the predator succeeds in capturing the prey, (2) the predator becomes exhausted, or (3) the duration of the episode exceeds a specified time limit. Six sensors provide the prey with information after each interaction with the predator: (1) the *time* into the episode, (2) the *range* of the predator from the prey, (3) the *bearing* of the pray relative to the predator (in "o'clock" terminology), (4) the *heading* of the predator relative to the prey (in degrees), (5) the *speed* of the predator, and (6) the *iff* flag, which indicates whether other agent is a friend or a foe (the EVADE problem can be generalized to consider several agents). Parameters for the Evade domain are given in Table 2.

Table 2. Problem Parameters

Parameters	
experiments = 5	rulebases = 2
gens = 50	safe = 100
popsize = 50	range_lo = 950
episodes = 20	range_hi = 1050
friends = 1	speed_lo = 650
foes = 1	speed_hi = 750
noise = 0.0 (or 0.1)	

In terms of the input files, the only difference between the problem presented here and the one presented in the *User's Guide* is that conditions in prey's attributes file are specified as structured attributes with a fuzzy matching option. The prey's attributes file is listed in Appendix A. The params file and the predator's attributes file remain unaltered. The predator uses SAMUEL's default rule matching mechanism.

In general, the code in a problem specific module must be consistent with the fuzzy matching of attributes. In the program module *evade.c*, the learning agent must store real-valued inputs to all of the sensors because the agent uses fuzzy attributes. Later on, in *samuel_find_match_atom()*, raw valued sensor inputs are bound to linguistic (symbolic) values. The non-learning agent (predator) stores real-valued inputs for numeric attributes (*time, range, bearing, heading, and speed*) and string values for symbolic attributes (*iff*), depending on the type of matching associated with each sensor's attribute. For this reason, the prey stores the current setting of the *iff* flag as a real value, whereas the predator stores this flag as a string to reflect the use of symbolic matching for this sensor.

The plot in Figure 4 is the output produced by SAMUEL's run-samuel script when learning agent (prey) uses fuzzy matching and the predator uses the system's default matching. This comparison plot shows the difference in the prey's performance when the agent's best strategies are tested with and without the presence of noise in the prey's sensors. The solid line indicates the performance without noise; the dashed line indicates the performance when the prey's sensors have a noise level of 0.10.

The structured attributes listed in Appendix A generally specify less resolution than the comparable non-structured attributes used to generate the output in the *User's Guide*. For example, in the fuzzy version, the time sensor only has five values: beginning (0, 0, 3, 4), near-beginning (3, 4, 7, 8), middle (7, 8, 11, 12), near-ending (11, 14, 15, 16), and ending (15, 16, 19, 20), whereas in the original version, the time sensor registers each of the 0 to 19 time steps. In order to consider the same resolution, a comparison run is made using the same structured attributes with the sensor range quantized into approximately equally-sized intervals over the leaf nodes. Figure 7 shows a comparable plot for the case where SAMUEL uses its default rule matching mechanism and non-fuzzy structured attributes. A comparison of Figures 4 and 5 show that, for noisy inputs to the sensors, SAMUEL performs better when using equally quantized ranges and the default rule matching mechanism. This difference in performance illustrates the difficulty of selecting good membership functions, especially for noisy inputs. Interpreting the intersection and union operators in fuzzy logic to be something other than the min and max operators, respectively, may produce better results (Harris, Moore, and Brown, 1993).

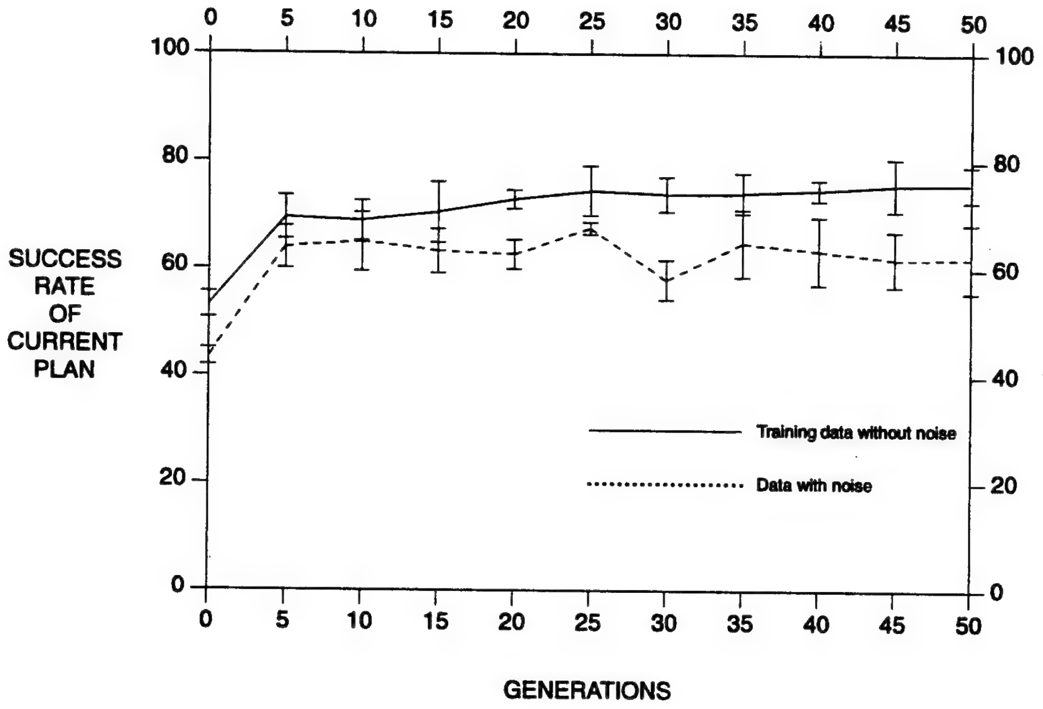

Fig. 4 - Comparison plot for EVADE problem when SAMUEL uses fuzzy rule matching.

Non-fuzzy systems may generate default outputs that creates instabilities whenever the system fails to recognize an input. In contrast, fuzzy rule systems typically produce outputs that are at least approximately correct, because membership functions overlap. When membership functions do not overlap, then the performance of the system can deteriorate so that the system exhibits some of the worse behaviors of non-fuzzy systems. For example, Figure 6 shows an example of the speed condition whose membership functions do not overlap. Notice that, even though the entire range for the speed condition is defined on input (0 to 1000), the membership functions do not cover the entire range. Figure 7 shows the dramatic deterioration in SAMUEL's performance when two of the linearly structured attributes, *range*, and *speed*, have non-overlapping membership functions. Clearly, the ranges of the membership functions have a significant effect on the performance of the system.

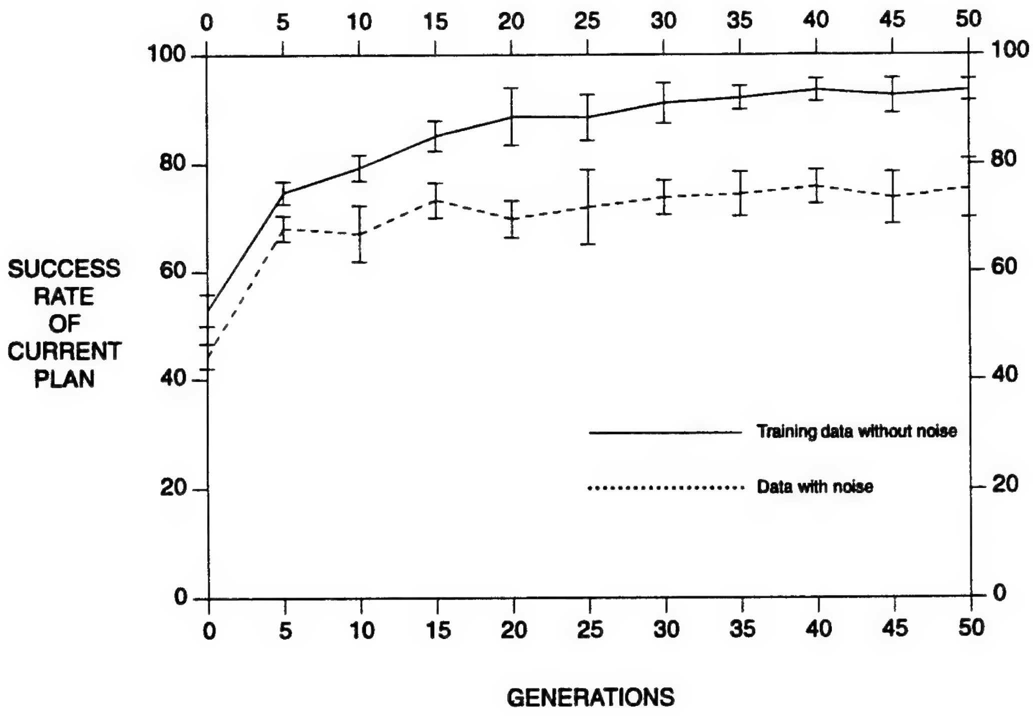

Fig. 5 - Comparison plot for EVADE problem when SAMUEL uses its default rule-matching mechanism

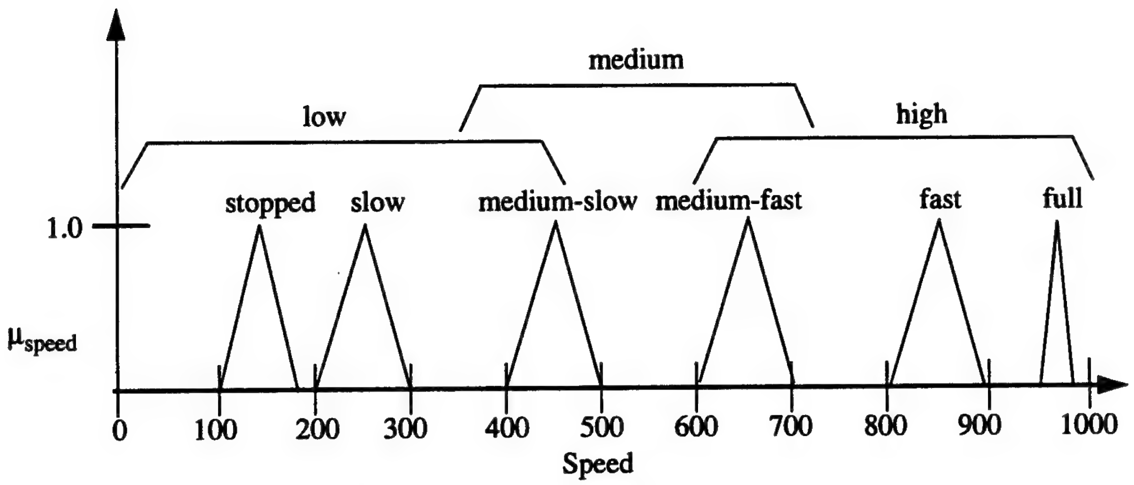

Fig. 6 - Membership functions for the speed condition where ranges do not overlap

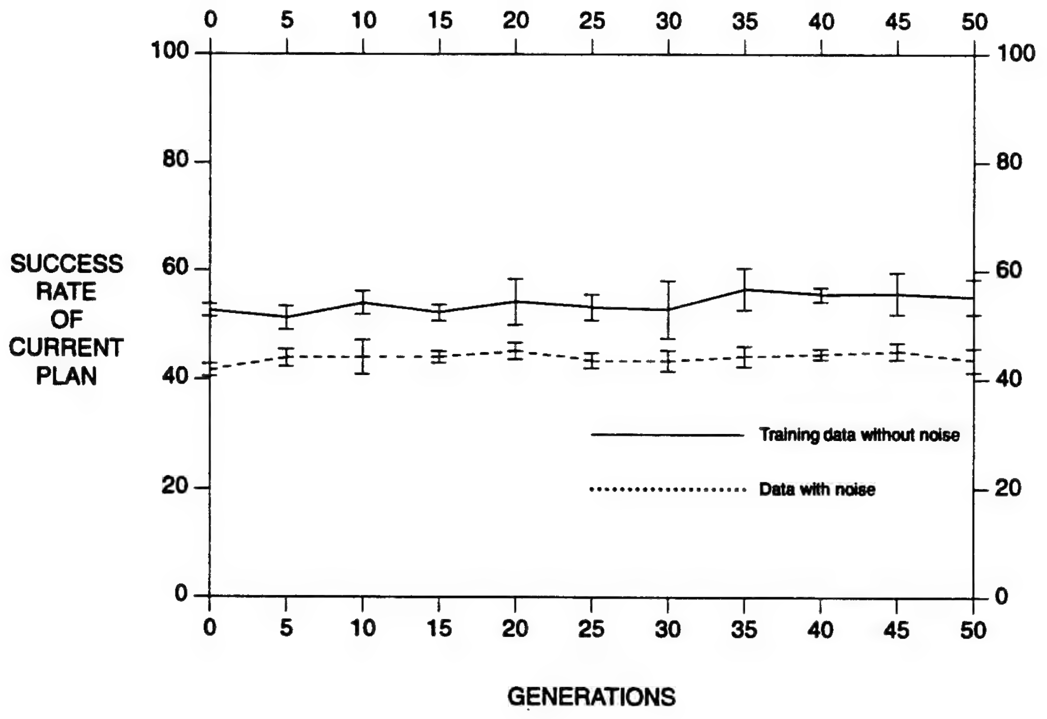

Fig. 7 - Comparison plot for EVADE problem when SAMUEL uses fuzzy rule matching.and the membership functions for the linear conditions do not overlap.

FUTURE ENHANCEMENTS

In the current implementation of the fuzzy logic in SAMUEL, membership functions are specified as linearly structured rulebase attributes that apply to all rules and all strategies, and this specification remains fixed over all of the generations. SAMUEL searches for strategies consisting of rules that express generalized conditions using the structure of these attributes.

A way of extending the use of fuzzy logic is to let the SAMUEL learn membership functions. New fuzzy operators would work on the quadtuples specifying the membership functions so that the system searches for membership functions as well as the level of generality in the rules. These operators would modify the low-partial, low-full, high-full, and high-partial values defining the membership function, similar to the way operators currently modify numeric attributes in SAMUEL. For example, if the quadtuple is (150, 300, 400, 450), a FUZZYMUTATE operator might change the low-full value from 300 to 400 so that the membership function in the new rule changes from a trapezoidal to a triangular shape. A FUZZYCREEP operator might increase or decrease the low-partial or high-partial values to change extent of the membership function's coverage.

To change the system so that there is a search over a set of membership functions, each strategy in the population would have a unique set of membership functions for the conditions. It is necessary to apply the same membership functions to all rules within a strategy in order to maintain semantic consistency. To ensure semantic consistency, the membership function definitions would be appended to the end of a strategy. New operators would be defined that operator on the leaf-values of the membership functions. Fuzzy versions of the SPECIALIZE, GENERALIZE, and COVER operators would only consider strategy-level information. A fuzzy version of the crossover operator would be needed to recombine the membership functions between two parent strategies.

Another enhancement is to include the defuzzying of control actions. Currently, SAMUEL uses only one control action value for each rule by default so that no defuzzying is required (i.e., the single_act parameter is set to one). If the control output of a rule were a fuzzy set, then some method for defuzzying the output would be required.

Finally, alternative fuzzy logic inference mechanisms need to be tested. One alternative is to use different operators than min and max for performing fuzzy intersections and fuzzy unions. Another alternative is to directly include membership values in the bidding process. A bid value would be a rule's strength multiplied its membership value. Since membership values measure the degree of partial match directly, no match lists would be necessary. This modification might increase the sensitivity of the inference mechanism to the membership values.

REFERENCES

John J. Grefenstette (1988). "Credit assignment in rule discovery system based on genetic algorithm," *Machine Learning* 3 (2/3), pp. 225-245.

John J. Grefenstette (1991). "Strategy acquisition with genetic algorithms", in *The Handbook of Genetic Algorithms*, L. Davis (Ed.), Boston: Van Nostrand Reinhold, 186-201.

John J. Grefenstette and Helen G. Cobb (1995). *User's Guide for SAMUEL, Version 4.* NRL Memorandum.

John J. Grefenstette, Connie Loggia Ramsey, and Alan C. Schultz (1990). "Learning sequential decision rules using simulation models and competition," *Machine Learning* 5(4), 355-381.

C. J. Harris, C. G. Moore, & M. Brown (1993). *Intelligent Control: Aspects of Fuzzy Logic and Neural Nets*. Singapore. World Scientific Publishing.

Charles L. Karr (1991). "Design of an Adaptive Fuzzy Logic Controller Using a Genetic Algorithm." *Proceedings of the Fourth International Conference on Genetic Algorithms* (ICGA-91). pp. 450-457. San Mateo, CA: Morgan Kaufmann.

Bart Kosko (1991). *Neural Networks and Fuzzy Systems: A Dynamic Systems Approach to Machine Intelligence.* Englewood Cliffs, NJ: Prentice Hall.

Chuen Chien Lee (1990). Fuzzy Logic in Control Systems: Fuzzy Logic Controller - Part I and Part II. *IEEE Transactions on Systems, Man and Cybernetics*, Vol. 20, No. 2, March, pp. 404-418, April, pp. 419-435.

Michael A. Lee and Hideyuki Takagi (1993). "Dynamic Control of Genetic Algorithms using Fuzzy Logic Techniques." *Proceedings of the Fifth International Conference on Genetic Algorithms* (ICGA-93). pp. 76-83. San Mateo, CA: Morgan Kaufmann.

Chuen-Tsai Sun and Jyh-Shing Jang (1993). "Using Genetic Algorithms in Structuring a Fuzzy Rulebase." *Proceedings of the Fifth International Conference on Genetic Algorithms* (ICGA-93). pp 655. San Mateo, CA: Morgan Kaufmann.

Hideyuki Takagi (1993). "Neural Network and Genetic Algorithm Techniques for Fuzzy Systems." *Proceedings of the World Congress on Neural Networks (WCNN '93)*. International Neural Network Society, pp. II-631 to II-634.

Philip Thrift (1991). "Fuzzy Logic Synthesis with Genetic Algorithms." *Proceedings of the Fourth International Conference on Genetic Algorithms* (ICGA-91). pp. 509-513. San Mateo, CA: Morgan Kaufmann.

Doug Walter (1993). *Proportional Navigation Guidance Using Fuzzy Logic*. Technical Report. Naval Air Warfare Center-Weapons Division, Chinalake, CA.

Lotfi A. Zadeh (1992). "The Calculus of Fuzzy If/Then Rules". *AI Expert*, March, pp. 23-27.

Lofti A. Zadeh (1973). "Outline of a New Approach to the Analysis of Complex Systems and Decision Processes", *IEEE Transactions on Systems, Man, and Cybernetics*, Vol. SMC-3, No. 1. January, pp. 28 - 44.

Appendix A

RULE ATTRIBUTE FILES FOR EVADE EXAMPLE

LEARNER'S FUZZY ATTRIBUTE FILE

```
conditions = 6
actions = 1
qualifiers = 0
compile = 0

condition 1 :
name = time
type = structured
match = fuzzy
low = 0
high = 20
order = linear
leaf-values = 5
beginning  ( 0, 0, 3, 4 )
near-beginning  ( 3, 4, 7, 8 )
middle  ( 7, 8, 11, 12 )
near-ending  ( 11, 14, 15, 16 )
ending  ( 15, 16, 19, 20 )
interior-values = 2
name = begin
children = 2
beginning near-beginning
name = end
children = 2
near-ending ending
exact = 0

condition 2 :
name = range
type = structured
match = fuzzy
low  = 0
high = 1500
order = linear
leaf-values = 5
left-far  ( 0, 0, 300, 400 )
left-close  ( 200, 300, 600, 700 )
center  ( 500, 600, 900, 1000 )
right-close  ( 800, 900, 1200, 1300 )
```

```
right-far ( 1100, 1200, 1500, 1500 )
interior-values = 2
name = left
children = 2
left-far left-close
name = right
children = 2
right-close right-far
exact = 0

condition 3 :
name = bearing
type = structured
match = fuzzy
low  = 1
high = 12
order = cyclic
leaf-values = 8
north-east  ( 1, 1, 2, 3 )
east  ( 2, 3, 3, 4 )
south-east  ( 3, 4, 5, 6 )
south  ( 5, 6, 6, 7 )
south-west  ( 6, 7, 8, 9 )
west  ( 8, 9, 9, 10 )
north-west  ( 9, 10, 11, 12 )
north  ( 11, 12, 12, 1 )
interior-values = 0
exact = 0

condition 4 :
name = heading
type = structured
match = fuzzy
low  = 0
high = 359
order = cyclic
leaf-values = 8
straight-in  ( 330, 0, 0, 30 )
in-left  ( 15, 45, 45, 75 )
str-left  ( 60, 90, 90, 120 )
out-left  ( 105, 135, 135, 165 )
str-out  ( 150, 180, 180, 210 )
out-right  ( 195, 225, 225, 255 )
str-right  ( 240, 270, 270, 300 )
in-right  ( 285, 315, 315, 345 )
interior-values = 4
name = left
children = 3
in-left str-left out-left
```

```
name = out
children = 3
out-left str-out out-right
name = right
children = 3
out-right str-right in-right
name = in
children = 3
in-right straight-in in-left
exact = 0

condition 5 :
name = speed
type = structured
match = fuzzy
low  = 0
high = 1000
order = linear
leaf-values = 6
stopped  ( 0, 0, 0, 100 )
slow  ( 100, 150, 250, 300 )
medium-slow  ( 300, 350, 450, 500 )
medium-fast  ( 500,  550, 650, 700 )
fast  ( 700, 750, 850, 900 )
full  ( 900, 1000, 1000, 1000 )
interior-values = 3
name = low
children = 3
stopped slow medium-slow
name = medium
children = 2
medium-slow medium-fast
name = high
children = 3
medium-fast fast full
exact = 0

condition 6 :
name = iff
type = structured
match = fuzzy
low = 0
high = 1
order = linear
leaf-values = 2
friend  ( 0, 0, 0, 0 )
foe ( 1, 1, 1, 1 )
interior-values = 0
exact = 1
```

```
action 1 :
name = turn
type = structured
match = numeric
low = -180
high = 180
order = linear
leaf-values = 9
hard-left
med-hard-left
med-left
soft-left
straight
soft-right
med-right
med-hard-right
hard-right
interior-values = 3
name = right
children = 4
soft-right
med-right
med-hard-right
hard-right
name = soft
children = 3
soft-left
straight
soft-right
name = left
children = 4
hard-left
med-hard-left
med-left
soft-left
```

ADVERSARY'S NON-FUZZY ATTRIBUTE FILE

```
conditions = 6
actions = 1
qualifiers = 0
compile = 1

condition 1 :
name = time
type = linear
low  =  0
high = 19
```

```
step = 1
exact = 0

condition 2 :
name = range
type = linear
low =   0
high = 1500
step = 100
exact = 0

condition 3 :
name = bearing
type = cyclic
low  =  1
high = 12
step = 1
exact = 0

condition 4 :
name = heading
type = structured
match = numeric
low  =  0
high = 315
order = cyclic
leaf-values = 8
straight-in
in-left
str-left
out-left
str-out
out-right
str-right
in-right
interior-values = 4
name = left
children = 3
in-left str-left out-left
name = out
children = 3
out-left str-out out-right
name = right
children = 3
out-right str-right in-right
name = in
children = 3
in-right straight-in in-left
exact = 0
```

```
condition 5 :
name = speed
type = structured
match = numeric
low = 0
high = 1000
order = linear
leaf-values = 6
stopped
slow
medium-slow
medium-fast
fast
full
interior-values = 3
name = low
children = 3
stopped slow medium-slow
name = medium
children = 2
medium-slow medium-fast
name = high
children = 3
medium-fast fast full
exact = 0

condition 6 :
name = iff
type = structured
match = symbolic
order = none
leaf-values = 2
friend foe
interior-values = 0
exact = 1

action 1 :
name = turn
type = structured
match = numeric
low = -90
high = 90
order = linear
leaf-values = 9
hard-left
med-hard-left
med-left
soft-left
```

```
straight
soft-right
med-right
med-hard-right
hard-right
interior-values = 3
name = right
children = 4
soft-right
med-right
med-hard-right
hard-right
name = soft
children = 3
soft-left
straight
soft-right
name = left
children = 4
hard-left
med-hard-left
med-left
soft-left
```

ADVERSARY'S FUZZY ATTRIBUTE FILE WITH NON-OVERLAPPING MEMBERSHIP FUNCTIONS FOR LINEAR ATTRIBUTES

```
conditions = 6
actions = 1
qualifiers = 0
compile = 1

condition 1:
name = time
type = structured
match = fuzzy
low = 0
high = 20
order = linear
leaf-values = 5
beginning ( 0,0,3,4 )
near-beginning ( 6,7,7,8 )
middle ( 10,11,11,12 )
near-ending ( 14,15,15,16 )
ending ( 16,19,19,20 )
interior-values = 2
name = begin
children = 2
```

```
beginning near-beginning
name = end
children = 2
near-ending ending
exact = 0

condition 2:
name = range
type = structured
match = fuzzy
low = 0
high = 1500
order =linear
leaf-values = 5
left-far ( 0,0,300,400 )
left-close ( 500,600,600,700 )
center ( 900,950,950,1000 )
right-close ( 1200,1250,1250,1300 )
right-far ( 1400,1450,1450,1500 )
interior-values = 2
name = left
children = 2
left-far left-close
name = right
children = 2
right-close right-far
exact = 0

condition 3:
name = bearing
type = structured
match = fuzzy
low = 1
high = 12
order = cyclic
leaf-values = 8
north-east ( 1,1,2,3)
east ( 2,3,3,4 )
south-east ( 3,4,5,6 )
south ( 5,6,6,7 )
south-west ( 6,7,8,9 )
west ( 8,9,9,10 )
north-west ( 9,10,11,12 )
north ( 11,12,12,1 )
interior-values = 0
exact = 0

condition 4:
name = heading
```

```
type = structured
match = fuzzy
low = 0
high = 359
order = cyclic
leaf-values = 8
straight-in ( 330,0,0,30 )
in-left ( 15,45,45,75 )
str-left ( 60,90,90,120 )
out-left ( 105,180,180,210 )
str-out ( 150,180,180,210 )
out-right ( 195,225,225,255 )
str-right ( 240,270,270,300 )
in-right ( 285,315,315,345 )
interior-values = 4
name = left
children = 3
in-left str-left out-left
name = out
children = 3
out-left str-out out-right
name = right
children = 3
out-right str-right in-right
name = in
children = 3
in-right straight-in in-left
exact = 0

condition 5:
name = speed
type = structured
match = fuzzy
low = 0
high = 1000
order = linear
leaf-values = 6
stopped ( 100,150,150,170 )
slow ( 200,250,250,300 )
medium-slow ( 400,450,450,500 )
medium-fast ( 600,650,650,700 )
fast ( 800,850,850,900 )
full ( 950,960,960,970 )
interior-values = 3
name = low
children = 3
stopped slow medium-slow
name = medium
children = 2
```

```
medium-slow medium-fast
name = high
children = 3
medium-fast fast full
exact = 0

condition 6:
name = iff
type = structured
match = fuzzy
low = 0
high = 1
order = linear
leaf-values = 2
friend ( 0,0,0,0 )
foe ( 1,1,1,1 )
interior-values = 0
exact = 1

action 1:
name = turn
type = structured
match = numeric
low = -180
high = 180
order = linear
leaf-values = 9
hard-left
med-hard-left
med-left
soft-left
straight
soft-right
med-right
med-hard-right
hard-right
interior-values = 3
name = right
children = 4
soft-right
med-right
med-hard-right
hard-right
name = soft
children = 3
soft-left
straight
soft-right
name = left
```

children = 4
hard-left
med-hard-left
med-left
soft-left

Appendix B

C CODE FOR FUZZY LOGIC IN SAMUEL

CODE SEGMENT FROM SAMUEL_CPS()

```
/* Action loop in samuel_cps() */

for (action = 0; action < AG->rulebase->actions; action++) {
        if (AG->rulebase->fuzzy)
                /* find set of matches for this action using fuzzy inference. *\/
                samuel_find_fuzzy_matches(AG, action);
        else
                /* find matching rules for this action */
                samuel_find_matches(AG, action);

        /* compute bids for each value associated with action */
        samuel_get_bids(AG, action);

        /* choose an action value */
        samuel_resolve_conflicts(AG, action);
}
```

SAMUEL_FIND_FUZZY_ MATCHES

```
#include "define.h"
extern int32 Debug;
extern int32 Gen;
extern int32 Agents;                    /* number of agents */
extern FILE *Stderr;
extern int32 Fuzzy_bins;

/********************** Local Declarations ************************/
typedef  struct rulenode  *RULENODEptr;
typedef struct rulenode {
        int32          rule;
        float64        value;
} RULENODE;

/******************************************************************************
* This procedure similar to samuel_find_matches(), except fuzzy logic is used to find the match lists.
* As in samuel_find_matches(), the strength of a rule is not considered at this point.  A set of lists is
* formed for each track for action a, where matchlist[Fuzzy_bins + 1][a] is a list of all rules having a
* membership of 1 (exact match); matchlist[0][a],  is the list holding no matches, and other matchlists
* are lists of rules, bined by ranges on membership values, depending on the number of Fuzzy_bins.
* Array matchout[j] indicates how many rules there are in each of the  match lists.
*/

int32 samuel_find_fuzzy_matches(AG, action)
        AGENT *AG;
        int32   action;
{
        int32 i;                        /* loop control */
        int32 j;                        /* loop control */
        int32 k;                        /* index of sensor value */
        int32 track;                    /* loop control over tracks */
        int32 conditions;               /* number of conditions */
        int32 nrules;                   /* number of rules */
        int32 count;                    /* number of rules in Membership List */
        int32 *matchcount;              /* how many rules match n sensors */
        float64 bin_width;              /* width of fuzzy bin */
        float64 bin_upper;              /* upper membership value of bin */
        float64 bin_lower;              /* lower membership value of bin */
        float64 membership_value;       /* possible membership value */
        float64 min_membership;         /* min membership value over the conditions */
        TRACK *TR;                      /* pointer to agent's track structure */
        RULEBASE_PTR rulebase;          /* pointer to agent's rule base */
        RULE *rule;                     /* pointer to agent's rules */
        static RULENODEptr Membership_List;         /* membership list */
                                        /* this list serves the same role as matchset */
        static int32 firstflag = 1;     /* allocate membership list only once */

        if (Debug) {
                fprintf(Stderr,"samuel_find_matches for agent %d\n", AG->id);
                fflush(Stderr);
        }

        /* Provide fast access to agent's data structures */
        rulebase = AG->rulebase;
        conditions = rulebase->conditions;
        nrules = rulebase->nrules;
        rule = rulebase->rule;

        /* Allocate membership list (array) */
        if (firstflag) {
                Membership_List = (RULENODEptr)
                        samuel_get_memory(MAX_RULES * SIZEOF(RULENODE));
                firstflag = 0;
        }

        if (Debug)
                printf("In samuel_find_fuzzy_matches\n");

        count = 0;
```

```
for (track = 0; track < Agents; track++) {
        if (AG->ignore[track])
                continue;

        TR = &(AG->track[track]);
        matchcount = TR->matchcount[action];

        for (j = 0; j < nrules; j++) {

                /* Skip rules that don't concern this action. */
                if (rule[j].act[action].lower == NULL_ACT)
                        continue;

                /* Find the minimum membership value over the conditions. */
                min_membership = 2.0;

                for (i = 0; (i < conditions) && (min_membership > 0.0); i++) {
                        if (rulebase->cond_attr[i].fuzzy) {
                                membership_value =
                                        samuel_fuzzy_match_atom(&rulebase->cond_attr[i],
                                                        rule[j].cond[i], TR->raw[i], &(TR->sensor[i]));

                                attribute = &AG->rulebase->cond_attr[i];

                                if (Debug) {
                                        fprintf(Stderr,"Cycle: %d  Episode: %d  Agent: %d  ",
                                                Cycle, Episode, AG->id);
                                        fprintf(Stderr,"Track: %d  Sensor: %16s  Raw Value: %f\n",
                                                track, attribute->name, TR->raw[i]);
                                        fprintf(Stderr, Value: %16s\n",
                                                samuel_attribute_value(attribute, TR->sensor[i]));
                                        fprintf(Stderr,"--------------------------------------------\n");
                                }
                        }
                        else {
                                membership_value = samuel_match_atom(&rulebase->cond_attr[i],
                                                        rule[j].cond[i], TR->sensor[i]);
                        }

                        /* Insist on a membership value of 1 for conditions where the match is
                                exact */
                        if (rulebase->cond_attr[i].exact)
                                if (membership_value != 1.0)
                                                membership_value = 0.0;

                        if (membership_value < min_membership)
                                min_membership = membership_value;
                }
                if (Debug) {
                        printf("min membership over conditions = %f\n", min_membership);
                        printf("************************************************\n");
                }

                /* Now put this rule membership array */
                Membership_List[count].rule = j;
                Membership_List[count++].value = min_membership;
        }

        if (Debug) {
                if (Gen > 0) {
                        printf("Membership Values: ");
                        for (i = 0; i < count; i++) {
                                printf(" %d (%f) ",
                                                Membership_List[i].rule, Membership_List[i].value);
                        }
                        printf("\n");
                }
        }

        /* Turn the membership linked list into the same format as matchlist[k][a] used in
        samuel_find_matches() and bid(). This way we can use bid(). Notice that the
        interpretation for index k isn't "the number of matching conditions" any more - it's
        based on the number of fuzzy bins (Fuzzy_bins). The matchlist[Fuzzy_bins + 1][a] is
        dedicated to exact matches (i.e., membership value of 1.0); the last matchlist[0][a] holds
        no matches (i.e., membership value = 0); other matchlists hold membership values over
        intervals in the range 1 to 0, exclusive of 1 and 0. */

        for (i = 0; i <= Fuzzy_bins + 1; i++)
                matchcount[i] = 0;

        for (i = 0; i < count; i++) {
                if (Membership_List[i].value == 1.0) {
                        TR->matchlist[Fuzzy_bins + 1][action][matchcount[Fuzzy_bins + 1]++] =
                                Membership_List[i].rule;
                }
        }

        bin_width = 1.0 / Fuzzy_bins;
        bin_upper = 1.0 - 1.0E-20;
        bin_lower = bin_upper - bin_width;
        for (k = Fuzzy_bins; k > 0; k--) {
                for (i = 0; i < count; i++) {
                        if ((bin_lower < Membership_List[i].value) &&
                                (Membership_List[i].value <= bin_upper)) {
                                        TR->matchlist[k][action][matchcount[k]++] =
                                                Membership_List[i].rule;
                        }
```

```
                    }
                    bin_upper -= bin_width;
                    bin_lower = bin_upper - bin_width;
               }

               for (i = 0; i < count; i++) {
                    if (Membership_List[i].value == 0.0)
                         TR->matchlist[0][action][matchcount[0]++] =
                              Membership_List[i].rule;
               }

               if (Debug) {
                     if (Gen > 0) {
                         for (i = Fuzzy_bins + 1; i >= 0; i--) {
                              printf("Matchlist %d:\n", i);
                              for (k = 0; k < matchcount[i]; k++)
                                   printf("list element %d : rule %d\n",
                                        k, TR->matchlist[i][action][k]);
                         }


                         printf("track: %d\n", track);
                         for (i = Fuzzy_bins + 1; i >= 0; i--) {
                              printf("There are %2d rules in match list %d.\n", matchcount[i], i);
                         }
                         printf("\n");
                    }
               }
          }
          return;
     }
```

SAMUEL_FUZZY_MATCH_ATOM

```
     /****************************************************************************
     *  Samuel_fuzzy_match_atom() is similar to match_atom(), but the routine returns membership
     *  values (float64) and not a binary result (int32).  Samuel_fuzzy_match_atom()  also sets the
     *  sensor's value. The function currently only works on structured fuzzy attributes.
     *  ***************************************************************************/

     float64 samuel_fuzzy_match_atom(attribute, atom, raw_value, n)
          ATTRIBUTE  *attribute;
          ATOM  atom;
          float64  raw_value;
          int32  *n;
     {
          int32 i;                   /* loop control on leaf nodes */
          int32 j;                   /* leaf node index */
          uint32 pattern;            /* pattern of condition */
          int32 list[MAX_NODES];     /* list of nodes in atom */
```

```
int32 count;                  /* number of leaf nodes */
float64 max_membership;       /* returned value - maximum membership over matching
                                    leaves */
float64 leaf_membership;      /* membership value at a leaf */

if (attribute->rightside == 1 && (atom.lower == NULL_ACT))
        return 0.0;

if (Debug) {
        printf("In samuel_fuzzy_match_atom: value = %f\n", raw_value);
}

switch (attribute->type) {
case STRUCTURED :
        max_membership = 0.0;
        *n = samuel_fuzzy_no_match(attribute, raw_value);
        if (*n == attribute->leaf_values) {

                if (Debug) {
                        printf("----sensor----: %16s\n", attribute->name);
                        printf("attribute high:  %f low:  %f\n", attribute->high, attribute->low);
                }

                /*      See if raw input values fall within the non-zero ranges of the membership
                        functions for the values of sensor i */

                pattern = ((uint32) atom.lower << SHIFT_SIZE) + atom.upper;

                leaf_membership = 0.0;
                for (i = 0; i < count; i++) {
                        j = list[i];
                        leaf_membership = samuel_fuzzy_leaf_match(attribute, j, raw_value);
                        if (leaf_membership > max_membership) {
                                max_membership = leaf_membership;
                                *n = j;
                        }

                        if (Debug)
                                printf("pattern element %d, leaf_membership = %f\n",
                                        j, leaf_membership);
                }
        }

        if (Debug) {
                printf("Node with best match for sensor %16s = %d\n", attribute->name, *n);
                printf("Num of leaf values = %d; max membership = %f\n",
                        attribute->leaf_values, max_membership);
        }
        return(max_membership);
```

```
            break;

        default:
            printf("fuzzy_match_atom: Help! Fuzzy flag set,");
            printf(" but the attribute is not structured.\n");
            abort();
            break;
        }
        return(-1.0);
    }
```

SAMUEL_FUZZY_NO_MATCH

```
    int32 samuel_fuzzy_no_match(attribute, raw_value)
        ATTRIBUTE *attribute;
        float64 raw_value;
    {
        static uint32 pick_seed = 987654321;
        ATT_NODE *cell;            /* current node */
        int32 i;
        int32 j;
        int32 iminus1;
        int32 return_value;
        float64 diff1;
        float64 diff2;
        float64 denom;
        float64 numer;

        switch (attribute->order) {

        case CYCLIC:
            for (j = 0; j < attribute->leaf_values; j++) {

                if (j == 0) {
                    iminus1 = attribute->leaf_values - 1;
                    i = j;
                }
                else {
                    iminus1 = j - 1;
                    i = j;
                }

                if (((&attribute->node[iminus1])->high_partial) <
                   (&attribute->node[i])->low_partial) {
                    diff1 = attribute->high - (&attribute->node[iminus1])->high_partial\;
                    diff2 = (&attribute->node[i])->low_partial - attribute->low;
                    if ((raw_value >= (&attribute->node[iminus1])->high_partial) &&
                       (raw_value <= attribute->high))
                          numer = raw_value;
                    else if ((raw_value >= attribute->low) &&
                    (raw_value <= (&attribute->node[i])->low_partial))
                    numer = diff1 + (raw_value - attribute->low);
                denom = (&attribute->node[iminus1])->high_partial + diff1 + diff2;

                if (numer / denom < 0.5)
                    return_value = iminus1;
                else
                    return_value = i;

                if (Debug) {
                    printf("In samuel_fuzzy_no_match, cyclic, prev < current.\n");
                    printf("raw_value: %f, prev high partial[%d]: %f\n",
                        raw_value, iminus1,
                        ((&attribute->node[iminus1])->high_partial));
                    printf("current low partial[%d]: %f\n", i,
                        (&attribute->node[i])->low_partial));
                    printf("return value: %d\n", return_value);
                }
                return return_value;
            }
            else if (((&attribute->node[iminus1])->high_partial) >
                    (&attribute->node[i])->low_partial) {
                if ((raw_value >= ((&attribute->node[iminus1])->high_partial)) &&
                    (raw_value <= ((&attribute->node[i])->low_partial))) {
                    if ((raw_value - ((&attribute->node[iminus1])->high_partial)) /
                        ((&attribute->node[i])->low_partial -
                         (&attribute->node[iminus1])->high_partial) < 0.5)
                        return_value = iminus1;
                    else
                        return_value = i;

                    if (Debug) {
                        printf("In samuel_fuzzy_no_match, cyclic, prev < current.\n");
                        printf("raw_value: %f, prev high partial[%d]: %f\n",
                            raw_value, iminus1,
                            ((&attribute->node[iminus1])->high_partial));
                        printf("current low partial[%d]: %f\n", i,
                            (&attribute->node[i])->low_partial));
                        printf("return value: %d\n", return_value);
                    }

                    return return_value;
                }
            }
        }

        return (attribute->leaf_values);
        break;
        case LINEAR:
            j = attribute->leaf_values - 1;
            for (i = 0; i < attribute->leaf_values; i++) {
                if ((i == 0) &&
                        (raw_value >= attribute->low) &&
                        (raw_value  < ((&attribute->node[0])->low_partial)))
                        return 0;
                else if ((i == j) &&
                        (raw_value > ((&attribute->node[j])->high_partial)) &&
                        (raw_value <= attribute->high))
                        return j;
                else if ((raw_value >= ((&attribute->node[i - 1])->high_partial)) &&
                         (raw_value <= ((&attribute->node[i])->low_partial))) {
                        if ((raw_value - ((&attribute->node[i - 1])->high_partial)) /
                            ((&attribute->node[i])->low_partial -
                            (&attribute->node[i - 1])->high_partial) < 0.5)
                            return (i - 1);
                        else
                            return i;
            }
        }

        return(attribute->leaf_values);
        break;

        default:
            printf("Help! Not a good type for fuzzy matching: %c for attribute %s\\n",
                        attribute->type, attribute->name);
            abort();
            break;
        }
    }
```

SAMUEL_FUZZY_LEAF_MATCH

```
    float64 samuel_fuzzy_leaf_match(attribute, j, raw_value)
        ATTRIBUTE *attribute;
        int32 j;
        float64 raw_value;
    {
        ATT_NODE *cell;             /* current node */
        float64 leaf_membership;    /* membership value at a leaf */
        float64 lowest;
        float64 highest;
        float64 tmp_high_partial;

        cell = &attribute->node[j];
        lowest = attribute->low;
        highest = attribute->high;
```

```
switch (attribute->order) {
case CYCLIC :
        /* low_full to high_full */
        if (cell->low_full > cell->high_full) {
                if (((cell->low_full <= raw_value) && (raw_value <= highest)) ||
                        ((lowest <= raw_value) && (raw_value <= cell->high_full)))
                        leaf_membership = 1.0;
        }
        if ((cell->low_full <= cell->high_full) &&
           (cell->low_full <= raw_value) && (raw_value <= cell->high_full))
                leaf_membership = 1.0;

        /* low_partial to low_full */
        else if (cell->low_partial > cell->low_full) {
                if ((cell->low_partial <= raw_value) && (raw_value <= highest))
                        leaf_membership = (highest - cell->low_partial > 0) ?
                                (raw_value - cell->low_partial) / (highest - cell->low_partial) : 0.0;
                else if ((lowest <= raw_value) && (raw_value <= cell->low_full))
                        leaf_membership = (cell->low_full - lowest > 0) ?
                                (raw_value - lowest) / (cell->low_full - lowest) : 0.0;
        }
        else if ((cell->low_partial <= cell->low_full) &&
                 (cell->low_partial <= raw_value) && (raw_value <= cell->low_full))
                        leaf_membership = (cell->low_full - cell->low_partial > 0) ?
                                (raw_value - cell->low_partial) /
                                (cell->low_full - cell->low_partial) : 0.0;

        /* high_full to high_partial */
        else if (cell->high_full > cell->high_partial) {
                if ((cell->high_full <= raw_value) && (raw_value >= highest)) {
                        tmp_high_partial = highest + (cell->high_partial - lowest);
                        leaf_membership = (tmp_high_partial - cell->high_full > 0) ?
                                (tmp_high_partial - raw_value) /
                                (tmp_high_partial - cell->high_f\ull) : 0.0;
                }
                else if ((lowest <= raw_value) && (raw_value <= cell->high_partial))
                        leaf_membership = (cell->high_partial - lowest > 0) ?
                                (cell->high_partial - raw_value) / (cell->high_partial - lowest) : 0.0;
        }
        else if ((cell->high_full <= cell->high_partial) &&
                (cell->high_full <= raw_value) && (raw_value <= cell->high_par\tial))
                        leaf_membership = (cell->high_partial - cell->high_full > 0) ?
                                (cell->high_partial - raw_value) /
                                (cell->high_partial - cell->high_full) : 0.0;
        else
           leaf_membership = 0.0;
        break;

case LINEAR :
```

```
            /* cell->low_full to cell->high_full */
            if ((cell->low_full <= raw_value) && (raw_value <= cell->high_full))
                leaf_membership = 1.0;
            /* cell->low_partial to cell->low_full */
            else if ((cell->low_partial <= raw_value) && (raw_value <= cell->low_full))
                leaf_membership = (cell->low_full - cell->low_partial > 0) ?
                    (raw_value - cell->low_partial) / (cell->low_full - cell->low_partial) : 0.0;
            /* cell->high_full to cell->high_partial */
            else if ((cell->high_full <= raw_value) && (raw_value <= cell->high_partial)) {
                leaf_membership = (cell->high_partial - cell->high_full > 0) ?
                    (cell->high_partial - raw_value) / (cell->high_partial - cell->high_full) : 0.0;
            }
            /* low to low partial and high_partial to high */
            else
                leaf_membership = 0.0;
            break;
        }
        return(leaf_membership);
}
```